Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра автоматизации систем вычислительных комплексов

Никифоров Никита Игоревич

Исследование применимости алгоритмов сжатия данных к таблицам классификации в сетевом процессорном устройстве

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
к. ф.-м. н., доцент
Д. Ю. Волканов

Москва, 2021

Аннотация

В данной работе рассматривается проблема недостатка памяти для классификации пакетов в рамках архитектуры сетевого процессорного устройства (СПУ). Под классификацией понимается процесс идентификации сетевого пакета по его заголовку. Для выполнения классификации требуются таблицы потоков, которые могут содержать до десятков тысяч правил, и поэтому занимаемый ими объём памяти может достигать до десятков мегабайт. Стадия классификация выполняется на СПУ, который представляет из себя специализированную интегральную микросхему. Рассматриваемый СПУ имеет конвейерную архитектуру, каждый конвейер состоит из восьми вычислительных блоков. Каждый вычислительный блок имеет доступ к устройству памяти объёмом 64K, в котором хранится программа обработки сетевых пакетов. Таблицы потоков представляются в виде программы обработки пакетов на языке ассемблера. Таким образом возникает задача разработки алгоритмов сжатия для применения к таблицам потоков. В данной работе мы рассматривали следующие алгоритмы сжатия: алгоритм оптимального кеширования, алгоритм рекурсивного отсечения и не специализированные алгоритмы сжатия. Экспериментальное исследование разработанных алгоритмов сжатия было проведено на имитационной модели сетевого процессора.

Содержание

Введение

В настоящее время активно развиваются технологии программно-конфигурируемых сетей (ПКС) [15]. Для работы ПКС требуются высокопроизводительные коммутаторы, которые выполняют функцию передачи данных. Возникает задача разработки программируемого сетевого процессорного устройства (СПУ) [14], являющегося основным функциональным элементом коммутаторов. В работе рассматривается коммутатор функционирующих под управлением протокола OpenFlow. Правила обработки пакетов в котором представляются в виде таблицы потоков. В данной работе рассматриваются только простые таблицы потоков. В СПУ таблицы потоков представляются в виде программы обработки заголовков сетевых пакетов.

СПУ представляет из себя интегральную микросхему. В рассматриваемом СПУ применяется конвейерная архитектура, а именно на каждый входной порт коммутатора СПУ содержит конвейер, состоящий из вычислительных блоков. Каждый вычислительный блок имеет доступ к устройству памяти в котором хранится программа обработки заголовков сетевых пакетов. Рассматриваемый СПУ имеет ограниченный объём доступной памяти, для хранения программы обработки заголовков сетевых пакетов.

Исходя из функций сетевого процессора, целесообразно рассматривать архитектуру, основанную на наборе конвейеров, которая позволяет с фиксированной задержкой обрабатывать каждый пакет. Конвейер в сетевом процессоре состоит из вычислительных блоков. В данной работе рассматривался этап классификации пакетов. Под классификацией понимается процесс идентификации сетевого пакета по его признакам, определяемыми текущим протоколом. Таблицы потоков необходимы для процесса классификации пакетов. Таблица потоков — это набор правил, определенных протоколом OpenFlow. OpenFlow — один из наиболее распространенных протоколов для управления сетевым коммутатором ПКС. В этой статье рассматривается OpenFlow версии 1.3 [4]. Каждое правило содержит поле признака, битовую строку, по которой пакет может быть идентифицирован, и набор действий, которые СП выполняет с этим пакетом.

Таким образом, для выполнения классификации сетевой процессор должен включать в себя ассоциативное устройство. Для реализации этого устройства естественным

будет использование ассоциативной памяти. Однако единственный контроллер ассоциативной памяти будет являться узким местом, так как к нему должны иметь доступ все вычислительные блоки всех конвейеров. Соответственно, возникает потребность усложнения архитектуры, например, путём добавления в неё нескольких контроллеров ассоциативной памяти. Чтобы избежать сложной организации памяти, в архитектуре можно отказаться от использования ассоциативной памяти. В таком случае одно из решений — совместить память команд и данных, и разместить память на кристалле сетевого процессора.

Современные таблицы потоков занимают до нескольких десятков мегабайтов памяти [11]. Поэтому возникает задача сжатия таблиц потоков, для использования рассматриваемого СП в коммутаторах ПКС.

Данная работа посвящена разработке алгоритмов сжатия данных [10] для применения в трансляторе таблиц потоков в рамках рассматриваемой архитектуры сетевого процессорного устройства.

1 Цели и задачи работы

Целью данной работы является исследование и разработка алгоритмов сжатия данных для применения к таблицам потоков OpenFlow. Для достижения поставленной цели необходимо было выполнить следующие задачи:

- Провести обзор существующих алгоритмов сжатия данных и способов их применения для сжатия таблиц потоков OpenFlow с целью выбора для применения в рассматриваемой архитектуре сетевого процессорного устройства.
- Провести анализ изменений, необходимых для применения в данном сетевом процессорном устройстве.
- Внести необходимые изменения в выбранные алгоритмы сжатия данных.
- Реализовать выбранные алгоритмы сжатия данных для проверки на эмуляторе сетевого процессорного устройства.
- Реализовать необходимые изменения эмулятора сетевого процессорного устройства, необходимые для проверки реализованных алгоритмов сжатия данных.
- Провести экспериментальное исследование реализованных алгоритмов сжатия данных.

2 Введение в предметную область

2.1 Протокол OpenFlow

В данной работе рассматривается протокол OpenFlow — один из наиболее распространённых протоколов для управления коммутаторов в ПКС сетях. В данной работе рассматривается версия OpenFlow 1.3 [4]. Для дальнейшего описания будем рассматривать устройство под управлением протокола OpenFlow в ПКС сети (OpenFlow коммутатора).

В спецификации протокола OpenFlow описываются абстракции: поток, признак, правило, действия, таблица потоков и т.д. С помощью этих абстракций описываются протоколы сетевого взаимодействия, а также описываются команды в которых контроллер управляет коммутатором. В протоколе OpenFlow отсутствуют ограничения на реализацию команд в коммутаторе.

Для процесса классификации коммутатором используются таблицы потоков, представляющие из себя набор правил. Каждое правило состоит из набора признаков и набора действий. Процесс классификации является процессом определения номера исходящего порта коммутатора, а также набора действий, которые необходимо применить к заголовку пакета перед его отправкой.

2.1.1 Таблица потоков

Поток — является некоторым множеством пакетов, каждый из которых идентифицируется определённым правилом. Для идентификации заголовка правилом используются признаки, которые делятся на две большие группы: признаки с точным значением и признаки с маскировкой значения.

- Признак с точным значением задаётся числом, в случае совпадения признака с полем заголовка пакета, которое соответствует данному признаку, пакет считается идентифицируемым данным правилом.

- Признак с маскировкой значения задаётся с использованием битовой маски, длина которой соответствует данному признаку.При использовании маски правило будет идентифицировать все пакеты по данному признаку, если определя-

емые маской биты поля в заголовке пакета совпадают с соответствующими битами значения, заданного в правиле. При использовании признака с маской значения может возникнуть неопределённость, а именно заголовок пакета может быть идентифицирован несколькими правилами. Для разрешения неопределённости используется приоритет, задаваемый для каждого правила. При возникновении неоднозначности заголовок пакета считается идентифицируемым правилом с большим приоритетом.

2.2 Архитектура аппаратного и программного обеспечения OpenFlow коммутатора

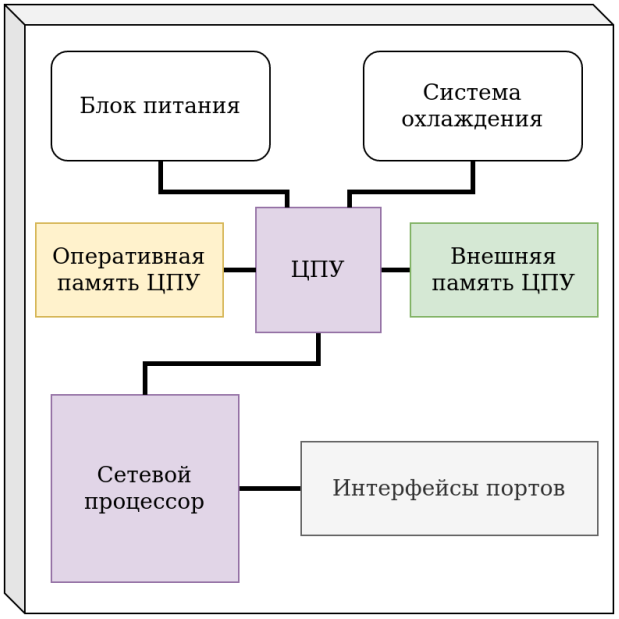


Рис. 1: Структурная схема коммутатора

Коммутатор, функционирующий под управлением протокола OpenFlow, включает в себя следующие основные компоненты (рис. 1):

- Сетевой процессор, выполняющий функцию передачи данных. Данное устройство получает пакеты через интерфейсы портов коммутатора, обрабатывает их, исполняя ранее загруженный машинный код, и затем отправляет необходимые пакеты через интерфейсы портов коммутатора.
- Центральное процессорное устройство (ЦПУ), выполняющее функцию интерфейса с контроллером ПКС сети, функцию обновления машинного кода сетевого

процессора, а также функцию управления питанием и охлаждением коммутатора.

- Блоки оперативной и внешней памяти для ЦПУ.
- Интерфейсы портов коммутатора.
- Блоки питания и система охлаждения.

Специальное программное обеспечение (СПО) выполняется в среде операционной системы на ЦПУ. СПО включает в себя следующие программные средства:

- агент протокола OpenFlow (коммуникация с контроллером и интерпретация его сообщений);
- система трансляции абстракций протокола OpenFlow в язык ассемблера сетевого процессора;
- транслятор языка ассемблера сетевого процессора в машинный код;
- драйверы сетевого процессора и других устройств коммутатора;
- служебное программное обеспечение для управления питанием, охлаждением и пр.

2.3 Архитектура сетевого процессора (RuNPU)

В рамках работы важны лишь некоторые особенности рассматриваемой архитектуры, а именно, отсутствие выделенного ассоциативного устройства памяти и конвейерная архитектура. Данные особенности непосредственно влияют на ограничения, предъявляемые к реализуемым структурам данных.

В сетевом процессоре используется конвейерная архитектура, каждый конвейер состоит из 10 вычислительных блоков. Вычислительный блок — это набор более низкоуровневых RISC ядер, которые в данной работе не рассматриваются. Каждый вычислительный блок имеет доступ к участку памяти, в котором располагаются микрокод и данные (рис. 2). Существует ограничение на количество тактов, которое один пакет может обрабатываться на вычислительном блоке, оно соответствует 25 тактам. Данное

ограничение обусловлено требованием к производительности сетевого процессора, а именно фиксированное время обработки одного пакета на сетевом процессоре. Также один вычислительный блок имеет доступ к 64 килобайтам памяти. Из-за особенностей микроархитектуры, отсутствует отдельная область памяти, в которой хранятся данные. Поэтому микрокод содержит в себе все данные, необходимые для классификации пакетов.

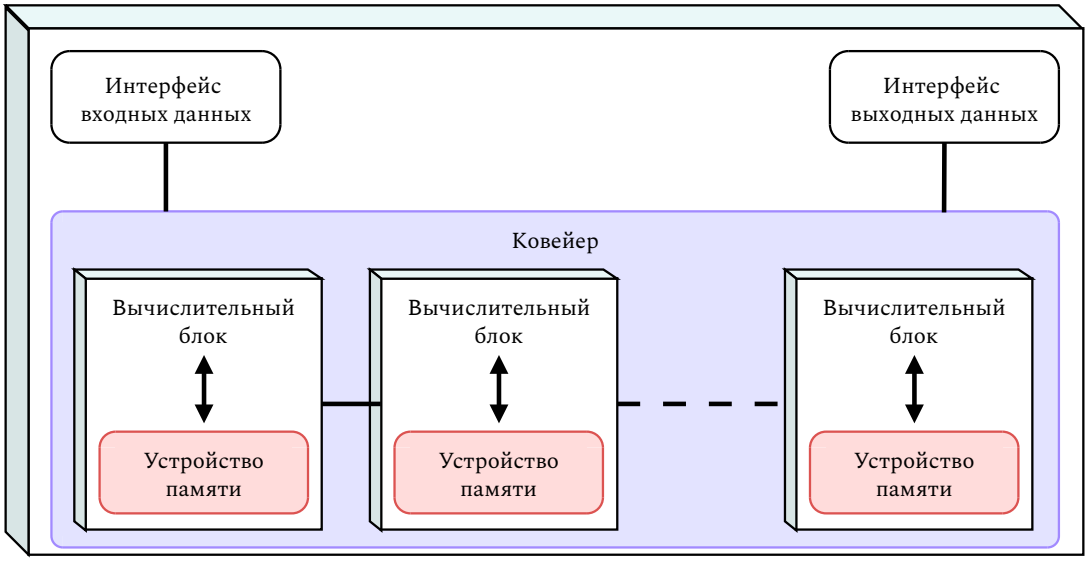


Рис. 2: Архитектура рассматриваемого сетевого процессора

2.4 Язык ассемблера сетевого процессора

Для описания программ обработки сетевых пакетов в рассматриваемой архитектуре сетевого процессора используется язык ассемблера. Вычислительный блок конвейера содержит единственный регистр общего назначения длиной 128 бит (регистр-аккумулятор), который выступает в качестве регистра-операнда и/или регистра-результата команды.

Память, доступ к которой осуществляется через команды, разделена на нескольких частей:

- область заголовка пакета;

- область метаданных (размер заголовка, номер порта, на который пришел пакет, маска выходных портов, пользовательские метаданные и др.).

Также вычислительный блок конвейера содержит регистр смещения. При выполнении команд, осуществляющих доступ в область памяти используется адрес равный сумме адреса, прописанного в команде, и значения регистра смещения.

Язык ассемблера сетевого процессора предназначен для описания программ, которые исполняются вычислительным блоком конвейера. В рассматриваемом языке присутствуют следующие классы инструкций:

- Инструкции для работы с регистром.
- Инструкции арифметических операций.
- Инструкции битовых операций.
- Инструкции для условного и безусловного перехода на метку.
- Инструкция записи в регистр выходного порта.

Также есть возможность использовать директивы имитационной модели $\#define$ и $\#include$, которые позволяют, например, задавать константы.

Язык ассемблера сетевого процессора также поддерживает специальные команды для вставки деревьев поиска. Директива tree_in включает дерево поиска по точному совпадению. В качестве параметра директивы указывается имя файла, строки которого содержат пары <значение, метка перехода>. Директива tree_lpm включает дерево поиска по совпадению наибольшего префикса. В качестве параметра директивы указывается имя файла, строки которого содержат тройки <начало диапазона, длина префикса, метка перехода>.

На программу на языке ассемблера рассматриваемого сетевого процессора накладываются следующие ограничения:

- Максимальная битовая длина сравниваемого значения в командах условного перехода и файлах деревьев поиска равна 64 битам.
- Метки переходов, содержащиеся в командах и файлах деревьев поиска, должны встречаться в программы после команды или директивы соответственно.

Это требование обусловлено архитектурой вычислительного конвейера сетевого процессора, в котором не допускается переход к командам, записанным в памяти по меньшему адресу, чем адрес текущей исполняемой команды.

Важной особенностью также является отсутствие отдельной области памяти для данных программы. Данные задаются в виде аргументов команд и являются неотделимой частью машинных команд.

3 Постановка задачи

3.1 Неформальная постановка задачи

Рассматривается коммутатор, функционирующий под управлением протокола OpenFlow, описанного в разделе 2.1. Коммутатор работает под управлением сетевого процессора (RuNPU), архитектура которого описана в разделе 2.3. Сетевой процессор работает под управлением программы на языке ассемблера, которая получается путём трансляции таблиц потоков OpenFlow

Пусть имеется таблица потоков OpenFlow, содержащая в себе правила с признаками с точным совпадением и маскирующие признаки. Имеется транслятор таблиц потоков, с помощью которого получается программа обработки сетевых пакетов на языке ассемблера. То есть классификацию полученного пакета по этой таблице потоков, то есть нахождение в таблице правила с наибольшим приоритетом, идентифицирующим обрабатываемый пакет. В данный момент программа на языке ассемблера при большом количестве правил в таблице потоков может занимать объём памяти в несколько раз больший, чем ограничения архитектуры (см. раздел 2.3).

Таким образом, возникает задача разработки алгоритмов сжатия таблиц потоков, для уменьшения объёма памяти занимаемого программ на языке ассемблера, получаемых из таблиц потоков путём их трансляции. При этом программа получаемая из сжатой таблицы потоков должна быть идентична программе получаемой из исходной таблице потоков.

3.2 Формальная постановка задачи

Введём формализацию OpenFlow таблиц. Упорядоченное множество всех рассматриваемых признаков в правилах обозначим $I = \{m_1, m_2, \ldots, m_k\}$. Каждый признак m_i из множества признаков I характеризуется битовой строкой, некоторой длины $m_i \in \{0, 1, *\}_i^W$, в данном случае символ $*$ обозначает любой бит. При этом, если $\exists m_i^j \in m_i$, такое, что $m_i^j = *$, то для $\forall m_i^k$, где $k > j$, то $m_i^k = *$. Длиной признака обозначим $len(m_i) = W_i$

Представим таблицу потоков в виде множества правил $R = \{r_1, r_2, \ldots, r_n\}$. С каждым правилом r_i связаны:

- номер i;
- приоритет $p_i \in Z_+$;
- вектор значений признаков $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$, где f_i^j соответствует значению признака $m_j \in I$.
- Набор действий, $A_i = \{a_1, a_2, \ldots, a_z\}$, которые определяют дальнейшие действия сетевого процессора над пакетом.

Будем говорить, что заголовок пакета x и его метаданные с вектором значений признаков $g = \{g^1, g^2, \ldots, g^k\}$ (далее $x \to g$), соответствуют правилу $r_i \in R$ с вектором значений признаков $f_i = \{f_i^1, f_i^2, \ldots, f_i^k\}$ и приоритетом p_i (правило $r_i \in R$ идентифицирует пакет с вектором значений признаков g), если:

1. вектор значений признаков g соответствует вектору значений признаков f_i, то есть $\forall g_i \in g, len(g_i) = len(f_i)$. И $\forall f_i^{lj} \in f_i^l$, $f_i^{lj} \in \{*, g^{lj}\}$, $l = \overline{1,k}$;
2. приоритет p_i максимален среди всех правил $r_j \in R$, для которых g соответствует вектору значений признаков f_j.

Множество R также должно удовлетворять следующему ограничению. Для любых двух правил $r_i, r_j \in R, r_i \neq r_j$, если их вектора значений пересекаются, то есть существует набор значений признаков, который соответствует векторам значений признаков обоих правил, то $p_i \neq p_j$. Например, правила с векторами значений признаков $f_i = \{110, 011, 1*\}$ и $f_j = \{11*, 011, 11\}$ должны иметь разный приоритет, так как набор значений признаков $g = \{110, 011, 11\}$ соответствует обоим правилам.

Введём функцию идентификации заголовка пакета $x \to g$ в таблице потоков R и обозначим её $R(x)$. Функция идентификации заголовка возвращает набор действий, соответствующий правилу, идентифицирующему заголовок пакета $x \to g$. Таким образом $R(x) = A_{r_i}$, где A_{r_i} набор действий правила $r_i \in R$.

Введём понятие аналогичности множеств R_1 и R_2. Множество R_1 аналогично множеству R_2, если для любого заголовка пакета, для которого существует идентифицирующее его правило $r_i \in R_1$, найдётся правило идентифицирующее его в множестве $r_j \in R_2$, при этом $A_i = A_j$.

Необходимо разработать алгоритм сжатия таблиц потоков, который будет переводить исходное множество $-$ R_1, соответствующее исходной таблице потоков, в новое множество R_2, которое соответствует новой таблице потоков.

1. Множество R_1 должно быть аналогично множеству R_2.

2. Мощность множества R_2 должна быть меньше либо равно мощности множества R_1.

Введём операцию последнего значащего бита признака $last(m_i) = j$, такое, что $m_i^j \in \{0, 1\}$ и $m_i^{(j+1)} = *$. Назовём правила $r_i \in R$ и $r_j \in R$ похожими, если для $\forall u \in len(f_i)$ верно, что $last(f_i^u) = last(f_j^u) = l$, при этом $f_i^{ul} \neq f_j^{ul}$, и $A_i = A_j$.

4 Обзор существующих алгоритмов сжатия

4.1 Цель и задачи обзора

Целью данного обзора является выбор алгоритмов сжатия для применения в сетевом процессорном устройстве. Необходимость применения алгоритмов сжатия обусловлена недостатком объёма памяти конвейера сетевого процессорного устройства.

В настоящем обзоре будут использоваться следующие критерии:

1. Отношение объёма памяти занимаемого таблицей классификации после применения алгоритма сжатия, к изначальному объёму памяти занимаемому таблицей классификации (Степень сжатия).

2. Оценка сложности сжатия.

3. Возможность использования без декомпрессии таблиц классификации — из-за ограничений рассматриваемого сетевого процессорного устройства сжатые таблицы классификации должны представляться как код ассемблера.

4. Использование внешней памяти — необходимость использования внешней памяти для использования сжатых таблиц классификации без декомпрессии.

Данные критерии обусловлены особенностями рассматриваемой архитектуры СПУ. Критерий **степень сжатия** необходим для оценки эффективности работы алгоритма сжатия. Критерий **оценки сложности сжатия** необходим для оценки накладных расходов на центральный процессор коммутатора при использовании рассматриваемого алгоритма сжатия. Критерий **возможности использования сжатых таблиц потоков без декомпрессии** обусловлен ограничениями архитектуры СПУ, а именно отсутствием адресуемой памяти в вычислительных блоках конвейера и ограниченным набором команд доступных СПУ (Раздел 2.4). Критерий **необходимости использования внешней памяти** обусловлен дополнительными накладными расходами при обработке сетевым процессорным устройством заголовков сетевых пакетов.

Для описания алгоритмов сжатия в обзоре будет использована терминология описанная в разделе 3.2.

4.2 Рассматриваемые алгоритмы сжатия

4.2.1 Распространённые алгоритмы

Под распространёнными алгоритмами сжатия будем понимать алгоритмы, которые сжатые данные представляют в бинарном виде [6]. Данные алгоритмы делятся на две большие группы:

1. Алгоритмы сжатия без потерь — наиболее простые алгоритмы сжатия, делятся на несколько категорий (рис. 3):
    - Статические: алгоритм Шенона-Фано, алгоритм Хаффмана, алгоритм арифметического кодирования.
    - Алгоритм RLE.
    - Алгоритм KWE.
    - Словарные и словарно-статистические алгоритмы: алгоритм LZ, алгоритм LZW.
2. Алгоритмы сжатия с потерями — особенность данных алгоритмов в том, что исходные данные не могут быть однозначно получены из сжатых. Данная особенность позволяет более хорошую степень сжатия, чем у алгоритмов сжатия без потерь. Также данная особенность ограничивает область использования таких алгоритмов и поэтому они не будут рассматриваться в рамках данной работы. Но можно упомянуть такие алгоритмы как:
    - Алгоритм JPEG.
    - Семейство алгоритмов H.26x.
    - Семейство алгоритмов MPEG.

Рассмотрим критерии для данного класса алгоритмов. Различные алгоритмы в данной группе имеют различную **степень сжатия**, которая колеблется от 0.3 до 0.8. Большинство рассматриваемых алгоритмов имеют квадратичную или кубическую **сложность сжатия** от объёма сжимаемых данных. Так как сжатые данные после применения алгоритмов из данной группы представляются в бинарном виде, **невозможно**

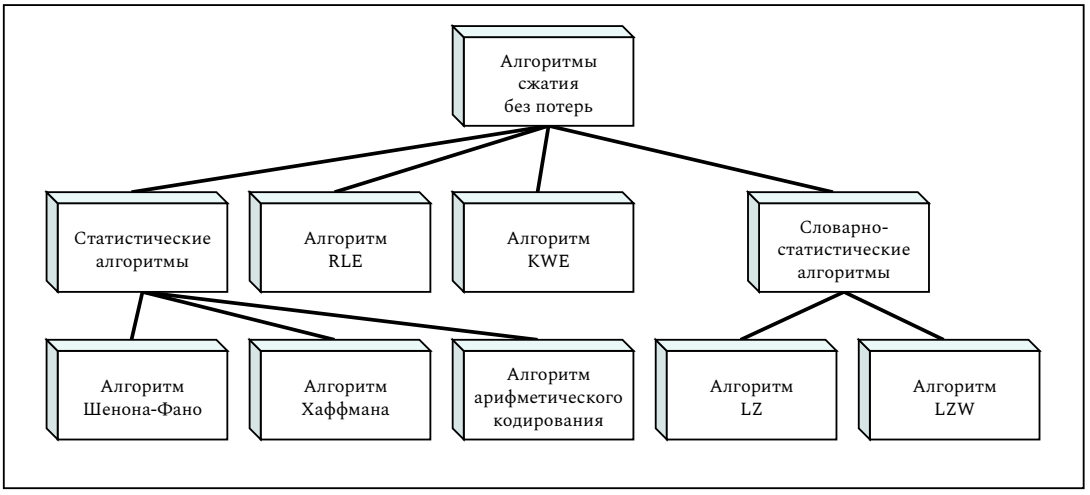


Рис. 3: Алгоритмы сжатия без потерь

использовать сжатые таблицы классификации без декомпрессии. И некоторые алгоритмы из рассматриваемой группы требуют доступ к **внешней памяти**.

4.2.2 Алгоритм оптимального кеширования

Данный алгоритм основан на построение дерева поиска правил относительно их частот [11], при этом, в данное дерево попадают только правила с определёнными частотами использования (далее популярность). Правила, не попавшие в основное дерево, хранятся на центральном процессоре коммутатора и доступ к ним осуществляется по запросу СП.

Для описания данного алгоритма потребуется ввести дополнительные обозначения. Введём понятие распределения заголовков пакетов P, где p_x обозначает вероятность получения пакета $x \to g = \{g^1, g^2, \ldots, g^k\}$. Также введём понятие коэффициента правильности $T_P(R_1, R_2)$, где R_1 и R_2 две различные таблицы потоков. Таким образом коэффициент правильности обозначает вероятность того, что заголовок пакета, согласно распределению P, будет в идентифицироваться правилами $r_1 \in R_1$ и $r_2 \in R_2$ и их наборы действий совпадают $A_1 = A_2, A_1 \in r_1, A_2 \in r_2$.

$$T_P(R_1, R_2) = \sum_{x \to g, R_1(x) = R_2(x)} p_x$$

Введём оптимальное значение коэффициента правильности для заданной таблицы

Таблица 1: Пример таблицы потоков.

Номер	**Признак**	**Действие над пакетом**	**Частота**
Правило 1	0xFFFFFFFF	Отправить пакет на порт 1	0.25
Правило 2	0xFFFEABAB	Отправить пакет на порт 2	0.2
Правило 3	0xFFFFCCCA	Отправить пакет на порт 3	0.15
Правило 4	0xDACA8415	Отправить пакет на порт 4	0.15
Правило 5	0xDEADC146	Отправить пакет на порт 1	0.1
Правило 6	0xBEAFAABA	Отправить пакет на порт 2	0.05
Правило 7	0xB2B3B7A9	Отправить пакет на порт 3	0.04
Правило 8	0xFFAADDEE	Отправить пакет на порт 4	0.03
Правило 9	0xBAEDEFFA	Отправить пакет на порт 1	0.015
Правило 10	0x9AF1FABF	Отправить пакет на порт 2	0.01
Правило 11	0x81ADBAEC	Отправить пакет на порт 3	0.005

потоков R, числа правил n и распределения заголовков P.

$$\zeta(n, R, P) = \max_{R_i, |R_i|<=n} T_P(R, R_i)$$

Таким образом алгоритму необходимо найти и построить таблицу потоков R_a, основанную на данной таблице потоков R с наименьшим количеством правил n_0 и максимальные оптимальным коэффициентом правильности $\zeta(n, R, P)$. Пусть p^i популярность (вероятность) выбора правила $r_i \in R$, в соответствие с распределением заголовков P. Пусть правила в таблице потоков R расположены в порядке не возрастания их популярности. Тогда:

$$\zeta(n, R, P) \geq \sum_{i \in [1,n]} p^i + 1 - \sum_{i \in [1,n_0]} p^i \geq n/n_0$$

Рассмотрим пример работы алгоритма для заданной таблицы потоков (Таблица 1). Правила сортируются по не возрастанию и в основное дерево попадают первые n правил, сумма частот которых не меньше 0.9. Остальные правила добавляются во второстепенное дерево. Результат разбиения изображён на рисунке 4.

Рассмотрим критерии для данного алгоритма: **Степень сжатия** данного алгоритма зависит от распределения частот использования префиксов, от 0.1 до 0.9. Данный алгоритм имеет квадратичную **сложность построения**. Сжатые таблицы классификации

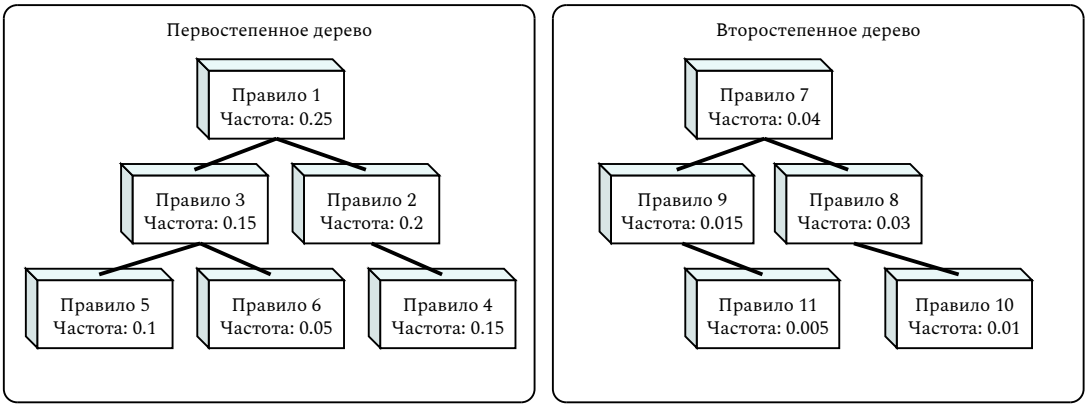


Рис. 4: Пример разбиения таблицы потоков на два дерева

возможно использовать без декомпрессии. Для работы данного алгоритма **требуется внешняя память**, так как часть префиксов, которая не попала в выборку наиболее часто используемых хранится в памяти центрального процессора коммутатора.

4.2.3 Алгоритм рекурсивного сокращения

Данный алгоритм основан на применении дерева HyperSplit [9], а сжатие производится за счёт удаления дублирующихся правил [2]. Данный алгоритм поддерживает добавление и удаление правил в таблице потоков.

Под дублирующимися правилами в таблице потоков понимаются следующие правила:

- правило, содержащиеся в вершине дублируется правилом в вершине, являющийся листом для данной вершины (частично дублирующиеся правило);
- правило, содержащиеся в вершине дублируется правилами во всех вершинах, являющихся листьями для данной вершины (полностью дублирующиеся правило).

Соответственно дублирующиеся правила перемещаются наверх дерева, что позволяет удалить полностью дублирующиеся правила.

Данный алгоритм рекурсивно использует NewHypersplit [5] [13] для удаления повторяющихся правил из дерева, которое строится в настоящее время. Затем удаленные повторяющиеся правила собираются в виде второй таблицы правил, называемой

рекурсивной таблицей, для построения второго дерева. Возможно, что во втором дереве все еще существуют повторяющиеся правила, и некоторые из них также удаляются и используются для построения третьего дерева. Этот процесс построения дерева выполняется рекурсивно до тех пор, пока в последнем дереве не будет дублированных правил.

Рассмотрим критерии для данного алгоритма: **Степень сжатия** данного алгоритма приблизительно равна 0.15. Данный алгоритм имеет **сложность построения** $n*log(n)$. Сжатые таблицы классификации **возможно использовать без декомпрессии**. Для работы данного алгоритма **не требуется внешняя память**, так как всё сжатие происходит в момент трансляции таблицы потоков в программу на языке ассемблера.

4.2.4 Алгоритм с использованием битовых векторов для представления таблиц потоков

Данный алгоритм основан на применение битовых строк для представления таблиц потоков [12]. А именно таблица потоков разбивается на несколько частей, в каждой из которых для всех битов префиксов записывается два значения: подходит ли данный префикс, если в искомой строке 1, подходит ли данный префикс, если в искомой строке 0. Таким образом поиск по таблицам потоков будет состоять из последовательного применения операции and. Все совпадающие поля делятся на $\frac{L}{s}$ sub-полей, где $(1 \leq s \leq L)$ обозначает длину подполя в битах. $K_j(j = 0, 1, ..., Ls–1)$ обозначает бит в подполе j. StrideBV уменьшает количество подходящих поисков, а также уменьшает задержку поиска.

Рассмотрим критерии для данного алгоритма: **Степень сжатия** данного алгоритма равна 0.5. Данный алгоритм имеет квадратичную **сложность построения**. Сжатые таблицы классификации **возможно использовать без декомпрессии**. Для работы данного алгоритма **требуется внешняя память**.

4.3 Сравнение алгоритмов сжатия

Для сравнения алгоритмов сжатия рассмотрим Таблицу 2. Сравнение будет проводится по следующим критериям:

У каждого рассмотренного алгоритма сжатия есть свои достоинства и недостатки, рассмотрим их:

Таблица 2: Сравнение алгоритмов сжатия.

Название	**Сложность построения**	**Степень сжатия**	**Внешняя память**	**Необходимость декомпрессии**
Оптимальное кеширование	$O(N^2)$	$0.1 \ldots 0.9$	да	нет
Рекурсивное удаления	$O(N*log(N))$	0.1	нет	нет
Использование битовых строк	$O(\frac{W}{K} * L)$	0.5	нет	нет
Распространённые алгоритмы	$O(K*\log_2 N)$	$0.1 \ldots 0.8$	нет	да

1. **Алгоритм оптимального кеширования** — имеет наибольший коэффициент сжатия, и быстро реализуем в рассматриваемой архитектуре сетевого процессорного устройства. Необходимость использования внешней памяти накладывает дополнительные расходы на обработку некоторых пакетов.

2. **Алгоритм рекурсивного удаления** — имеет наименьший коэффициент сжатия, реализуем сложнее, чем алгоритм оптимального кеширования. При этом данный алгоритм не требует использования внешней памяти.

3. **Алгоритм с использованием битовых строк** — имеет средний коэффициент сжатия, но при этом трудно реализуем в рассматриваемой архитектуре сетевого процессорного устройства.

4. **Распространённые алгоритмы сжатия** — в среднем имеют хорошие коэффициенты сжатия, но при этом требуется декомпрессия данных.

Таким образом, на основе обзора для дальнейшей реализации были выбраны два алгоритма сжатия данных. Алгоритм оптимального кеширования и алгоритм с использованием битовых строк.

4.4 Выводы

В данной главе был проведён обзор существующих алгоритмов сжатия, а именно: распространённые алгоритмы сжатия, алгоритм оптимального кеширования, алгоритм рекурсивного удаления и алгоритм с использованием битовых строк.

5 Эмулятор сетевого процессора

Эмулятор сетевого процессора — программа написанная на языке программирования $python3$, которая позволяет эмулировать работу сетевого процессора, а именно получать пакеты на любой из 24-х портов, обрабатывать пакет, получая информацию о любой стадии обработки, и отправлять пакет на выходные порты. Эмулятор сетевого процессора позволяет оценить количество тактов затраченных на обработку пакета, а также объём памяти затраченный на структуру данных.

Работа эмулятора сетевого процессора состоит из следующих шагов:

1. **Макрогенерация.** Производится замена констант, объявленных с помощью директив define, на их числовые значения. Происходит генерация деревьев поиска из значений, записанных в файлах деревьев поиска. При этом директивы включения деревьев поиска заменяются на соответствующие наборы команд и меток перехода.

2. **Запись программы во внутреннее представление.** Команды полученной на предыдущем шаге программы переводятся во внутреннее представление программы имитационной модели, моделирующее адресное пространство вычислительного конвейера.

3. **Обработка входных пакетов.** Происходит последовательная обработка входных пакетов с записью выходных пакетов. Также на этом шаге в режиме отладки возможна запись информации о последовательности выполнения команд.

4. **Вывод статистической информации.** По результатам обработки входных пакетов выводится количество обработанных пакетов, среднее число тактов, затраченное на обработку одного пакета, объем занимаемой программой памяти, данные по энергопотреблению и др.

5.1 Описание программных средств эмулятора сетевого процессора

Передача пакетов на порты сетевого процессора осуществляется с помощью файлов *.pcap* [3], в которых можно описать поступление пакетов на каждый порт в определённые моменты времени. В данной работе, не уменьшая общности, будет рассматри-

ваться только один порт эмулятора сетевого процессора. Так как для каждого входного порта используется соответствующий конвейер. Изначально, в конвейере эмулятора сетевого процессора был реализован один вычислительный блок. Для проведения экспериментального исследования в данной работе, необходимо внести изменения в эмулятор сетевого процессора, а именно увеличить количество вычислительных блоков в конвейере до 10.

5.2 Программные средства разработанные для проведения экспериментального исследования

Для проведения экспериментального исследования в рамках данной работы, необходимо разработать генератор сетевого трафика, который должен удовлетворять следующим требованиям:

- Возможность генерации L2 и L3 трафика.
- Возможность использовать наперёд заданную базу префиксов или mac-адресов для генерации трафика.
- Возможность использовать для генерации трафика заданное количество случайных префиксов или mac-адресов.
- Возможность сохранить базу использованных префиксов или mac-адресов.
- Возможность задать распределения времени поступления пакетов.

Для реализации будет использовать язык программирования $python3$, так как существует открытая библиотека *scapy*, которая позволяет работать с L2 и L3 трафиком, а так же позволяет сохранять сгенерированные пакеты в *.pcap* файлы, что необходимо, для дальнейшего использования в эмуляторе сетевого процессора.

6 Система трансляции таблиц потоков в язык ассемблера сетевого процессорного устройства

В данной главе приводится описание системы трансляции таблиц потоков OpenFlow [7]. Также будет представлено описание структур данных, использующихся для промежуточного представления таблиц потоков, и разработанных алгоритмов сжатия таблиц потоков и трансляции сжатых таблиц потоков в язык ассемблера СПУ.

6.1 Структуры данных

6.1.1 Структура данных для представления таблиц потоков в виде дерева

В системе трансляции для представлении таблицы потоков с набором правил R используется дерево с помеченными вершинами и дугами. С каждой вершиной дерева, кроме вершин-листьев, связаны следующие значения:

- признак из множества рассматриваемых признаков;
- подмножество набора правил R.

Структура данных строится по следующим правилам, где v — вершина дерева, которой соответствует признак m и подмножество правил $S \subset R$:

1. корню дерева соответствует всё множество правил R;
2. если M — множество всевозможных значений признака m в правилах из подмножества S, то для каждого значения $f \in M$ у вершины v существует потомок, к которому ведёт дуга с пометкой f.
3. если вершины u — потомок вершины v в которую ведёт дуга с пометкой f, то подмножество правил вершины u состоит только из правил в S, у которых значение признака m равно f.

Описанная структура данных позволяет выполнять поиск идентифицирующего обрабатываемый пакет правила в таблице потоков.

6.1.2 Структура данных для представления таблиц потоков в виде АВЛ дерева

В прошлой работе [8] была разработана структура данных для представления таблиц классификации в виде АВЛ дерева. Аналогично таблицу потоков можно также представить в виде АВЛ дерева, с каждой вершиной АВЛ дерева связаны следующие значения:

- скалярное значение соответствующее набору признаков $I = m_1, m_2, \ldots, m_k$, вычисляемое по расширенному алгоритму представления префиксов как скалярных величин.
- подмножество набора правил R, соответствующих данному набору признаков.

6.1.3 Структура данных для промежуточного представления таблицы потоков

Для применения алгоритмов сжатия необходимо промежуточное представление таблицы потоков. А именно в дополнение к существующим структурам данных к каждой вершине необходимо добавить следующие значения:

- Величина $P(S)$ соответствующая сумме частот использования каждого правила из подмножества $S \subset R$. $P(S) = \sum_{r_i \in S} P(r_i)$.
- Величина $P(N)$ соответствующая сумме частот использования потомков данной вершины. $P(N) = \sum_{f_i \in M} P(N(f_i))$. Если у данной вершины нет потоков, то $P(N) = P(S)$.

6.2 Модуль генерации ассемблерного кода

В данной работе в трансляторе таблиц потоков интересен модуль генерации ассемблерного кода. В нём реализован алгоритм трансляции структуры данных в программу на языке ассемблера сетевого процессора. Программа строится на основе одного из представлений таблицы потоков, используя соответствующий ему метод. Выходными данными модуля является набор файлов, включающий в себя основной файл программы с кодом на языке ассемблера и файлы деревьев поиска, встречающихся в директивах основного файла программы.

В данном модуле выделяются следующие внутренние модули:

- *Генератор ассемблерного кода* предоставляет интерфейс модуля генерации ассемблерного кода.
- *Модуль прямого способа трансляции* реализует алгоритм трансляции прямым способом посредством обхода вершин дерева, полученного модулем представления таблицы потоков в виде дерева.
- *Модуль способа трансляции с кодированием дуг* реализует алгоритм трансляции способом с кодированием дуг посредством обхода таблиц, полученных модулем представления таблицы потоков в виде уровней дерева.
- *Генератор команд* содержит общие функции для двух способов трансляции, например, функции для добавления записей в файлы деревьев поиска или функцию, создающую код на языке ассемблера, для загрузки значения поля заголовка пакета в регистр аккумулятор.

6.3 Алгоритмы сжатия

6.3.1 Использованные обозначения

Пусть $node_1$, $node_2$ — вершины дерева, $value$ — некоторое значение признака. Введем следующие обозначения:

- $Tree.root$ — корневая вершина дерева $Tree$.
- $node_1(value)$ — потомок вершины $node_1$, связанный с ней дугой с пометкой $value$.
- $node_1.rules$ — множество правил, соответствующее вершине $node_1$.
- $node_1.edges$ — множество пометок дуг, исходящих из вершины $node_1$.
- $copy(node_1, val, node_2)$ — процедура, которая добавляет к вершине $node_1$ потомка с дугой, помеченной val, копируя дерево, которое образует вершина $node_2$.
- $remove(node_1, val)$ — процедура, которая удаляет из вершины $node_1$, поддерево с дугой помеченной val.
- $equals(node_1, node_2)$ — функция, которая возвращает $true$, если деревья, образованные вершинами $node_1$ и $node2$ совпадают, иначе возвращает $false$. При сравнении учитываются связанные с вершинами множества правил и пометки дуг.

- $isleaf(node_1)$ — функция, которая возвращает $true$, если $node_1$ — лист дерева, иначе возвращает $false$.

6.3.2 Алгоритм предварительной оптимизации таблицы потоков

В данном разделе будет описан алгоритм предварительной оптимизации таблиц потоков. Для начала необходимо ввести операцию **Same**, которая на вход получает две вершины дерева и возвращает множество правил, которые были получены путём объединения похожих правил. Для описания алгоритма предварительной оптимизации

```
1 procedure Same(node_1, node_2):
2     rules = {}
3     for all rule_1 in node_1.rules do
4         for all rule_2 in node_2.rules do
5             if same(rule_1, rule_2) then
6                 rules += {rule_1 union rule_2}
7             endif
8     return rules
```

таблиц потоков необходимо ввести операцию **Optimise**, которая на вход получает корень дерева исходной таблицы потоков, а затем рекурсивно объединяет все похожие правила с помощью процедуры **Same**. После применения процедуры **Optimise** необ-

```
1 procedure Optimise(node):
2     if not isleaf(node) then
3         for all val_1 in node.edges do
4             for all val_2 in node.edges do
5                 if val_1 not equal val_2 then
6                     node.rules += Same(node(val_1), node(val_2))
7                 endif
8         for all val in node.edges do
9             Optimise(node(val))
```

ходимо удалить все листовые вершины, для этого опишем процедуру **Remove**, которая на вход получает корень дерева, а затем рекурсивно удаляет все листовые вершины в которых не осталось правил. Тогда полный алгоритм предварительной оптимизации можно описать последовательным применением процедур **Optimize** и **Remove**.

```
procedure Remove(node):
    if not isleaf(node) then
        for all val in node.edges do
            if isleaf(node(val)) and node(val).rules equal {} then
                remove(node, val)
            else
                Remove(node(val))
            endif
```

6.3.3 Алгоритм оптимального кеширования

В этом разделе представлен адаптированный алгоритм оптимального кеширования. Основной его частью является операция разделения исходной таблицы потоков на две. Для применения дальнейшего алгоритма необходимо построить изначальное дерево с дополнительными полями в узлах.

- Получить набор вершин дерева.
- Отсортировать этот набор в невозрастающем порядке сумм вероятностей вершин.
- Создать второй набор вершин, из которого в последствии будет строится дерево.
- Создать счетчик, хранящий сумму вероятностей вершин во втором наборе вершин.
- Получить первую вершину с максимальной суммой вероятностей.
- Увеличить счетчик на его вероятность.
- Добавить эту вершину во второй набор вершин и удалить из первого.
- Повторять последние три операции, пока счетчик меньше 0.95.
- Построить дерево из второго набора вершин.

После выполнения этих операций мы получаем два набора вершин, первый из которых отвечает за второстепенное дерево, а второй за первостепенное. Первостепенное дерево преобразуется в программу на языке ассемблера, которая загружается в СПУ.

Подробнее опишем алгоритмы, необходимые для реализации алгоритма оптимального кеширования. Для начала опишем процедуру **GetList**, которая на вход получает корневую вершину дерева $Tree.root$. На выходе мы получаем список вершин в которых есть правила.

```
procedure GetList(node):
    nodes = {}
    if node.rules not equal {} then
        nodes += node
    endif
    for all val in node.edges do
        nodes += GetList(node(val))

    return nodes
```

Также необходима процедура выбора первостепенных вершин **GetBest**, которая на вход получает отсортированный список вершин, и возвращает список первостепенных вершин.

```
procedure GetBest(nodes):
    counter = 0.0
    best_nodes = {}
    for all node in nodes do
        if counter is greater 0.95 then
            break
        endif
        counter += node.prob
        best_nodes += node
    return best_nodes
```

7 Экспериментальное исследование реализованных алгоритмов сжатия

При проведении экспериментального исследования ставились следующие цели:

- Оценка степени сжатия программы на языке ассемблера сетевого процессорного устройства, на различных данных.
- Оценка времени обновления таблиц потоков при использовании алгоритмов сжатия, на различных данных.

7.1 Методика экспериментального исследования

Для оценки параметров необходимо исследовать программу на языке ассемблера, получаемую при использовании системы трансляции с алгоритмами сжатия и без. Для каждой программы, с помощью эмулятора сетевого процессорного устройства, будут исследоваться следующие параметры:

- Объём памяти занимаемой программой при обработке пакетов на эмуляторе сетевого процессорного устройства.
- Среднее время обработки пакета в тактах сетевого процессорного устройства.

Для проведения экспериментального исследования, необходимо последовательно выполнять следующие действия для каждого набора входных данных:

1. Выбрать таблицу потоков для данного эксперимента.
2. Провести трансляцию выбранной таблицы потоков в программу на языке ассемблера:
   - без использования алгоритмов сжатия, обычное дерево;
   - без использования алгоритмов сжатия, с АВЛ деревом;
   - с использованием разработанных алгоритмов сжатия.
3. Провести эмуляцию работы сетевого процессорного устройства с полученными программами на языке ассемблера.
4. Провести оценку результатов полученных в данном эксперименте.

7.1.1 Данные экспериментального исследования

Для проведения экспериментального исследования мы будем использовать несколько вариантов таблиц потоков [1]. В этом разделе представлены шаблоны таблиц потоков, которые будут использованы для экспериментального исследования.

- Первый шаблон — шаблон правила таблицы потока содержит значения трех атрибутов: номер входного порта, MAC-адрес назначения и MAC-адрес источника.
- Второй шаблон — схема правила таблицы потока содержит значения двух атрибутов: Адрес назначения IPv4 и адрес источника IPv4.
- Третий шаблон — таблица правил содержит пять атрибутов: номер входного порта, MAC-адрес назначения, ID VLAN, ID заголовка L3-уровня (EtherType) и IPv4-адрес назначения.

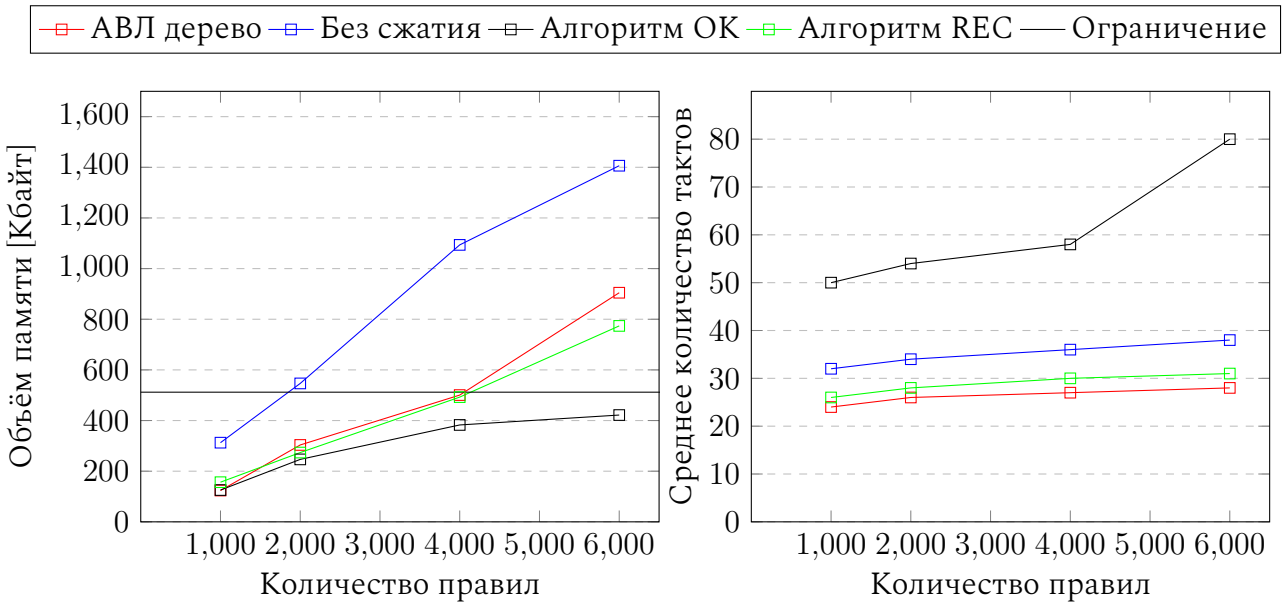
Ниже представлен пример входной таблицы данных:

```
{SRC_MAC , DST_MAC , INSTR}
{SRC_MAC , DST_MAC , INSTR}
+---+-----------+-----------+--------------+
| 1 | :12       | :10:1     | goto_table 1 |
+---+-----------+-----------+--------------+
| 1 | :23:45    | :20       | goto_table 1 |
+---+-----------+-----------+--------------+
| 1 | :0        | :1        | goto_table 1 |
+---+-----------+-----------+--------------+
```

7.2 Результаты экспериментального исследования

Для исследования характеристик программ, получаемых с помощью транслятора таблиц потоков с внедрёнными алгоритмами сжатия таблиц потоков, был проведен ряд замеров для таблиц, содержащих от 1000 до 6000 правил. В замерах участвовали таблицы потоков, содержащие только правила одного из описанных в разделе 7.1.1 шаблонов.

После оценки объёма памяти, занимаемого получаемой программой, были получены следующие результаты:



(a) Зависимость затраченного объёма памяти от количества правил

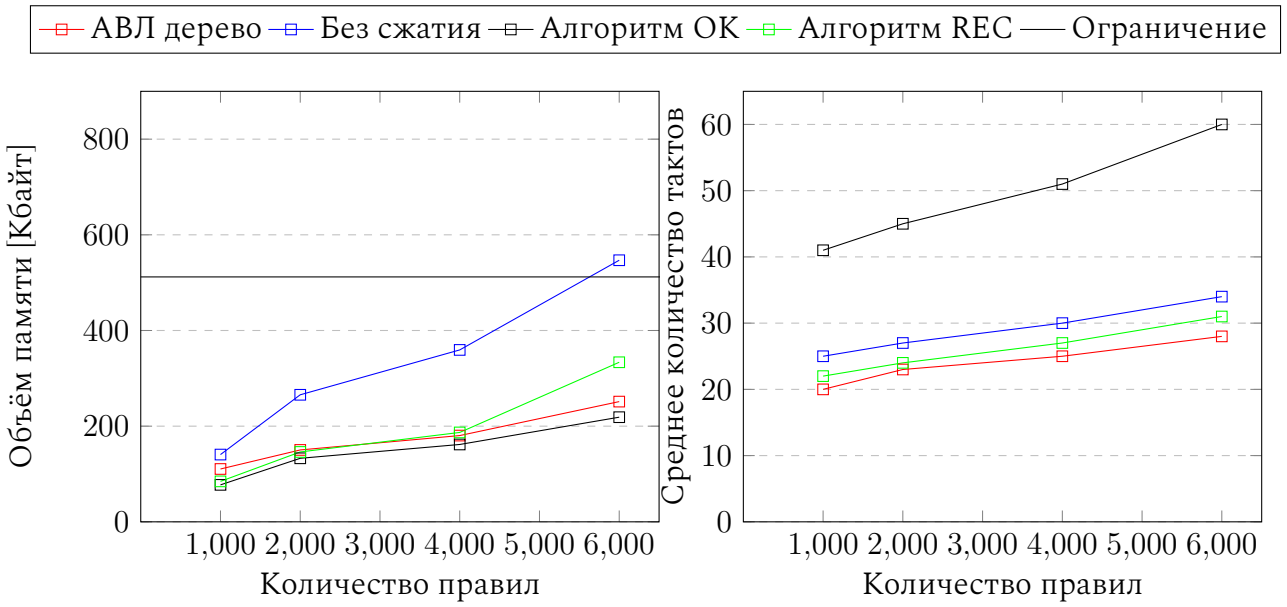
(b) Зависимость количества тактов памяти от количества правил

Рис. 5: Шаблон 1

Для всех шаблонов были получены ожидаемые зависимости использованного объёма памяти программой от количества правил в таблицах потоков OpenFlow. Для алгоритма рекурсивного отсечения получена линейная зависимость, для алгоритма оптимального кеширования получена логарифмическая зависимость. Данные зависимости изображены на рисунках 5a, 6a, 7a.

Наибольший объем памяти требовался для программ, полученных из таблиц потоков, содержащих правила *Шаблона 1* (рисунки 5a), несмотря на то, что такие правила содержат три признака в отличие от пяти и двух признаков у остальных шаблонов. Это можно объяснить наличием признаков в правилах, которые могут принимать как точное значение, так и значение $*$, что увеличивает размер структуры данных, соответствующей данной таблице потоков. Для алгоритма рекурсивного отсечения получено уменьшение объёма памяти, используемого программой на языке ассемблера, в 2 раза. Для алгоритма оптимального кеширования получено уменьшение объёма памяти, используемого программой на языке ассемблера, в 3 раза.

Наименьший объем памяти требовался для программ, полученных из таблиц потоков, содержащих правила *Шаблона 2* и *Шаблона 3* (рисунки 6a, 7a), Это можно объяснить наличием признаков в правилах, которые могут принимать только точные значения,



(a) Зависимость затраченного объёма памяти от количества правил

(b) Зависимость количества тактов памяти от количества правил

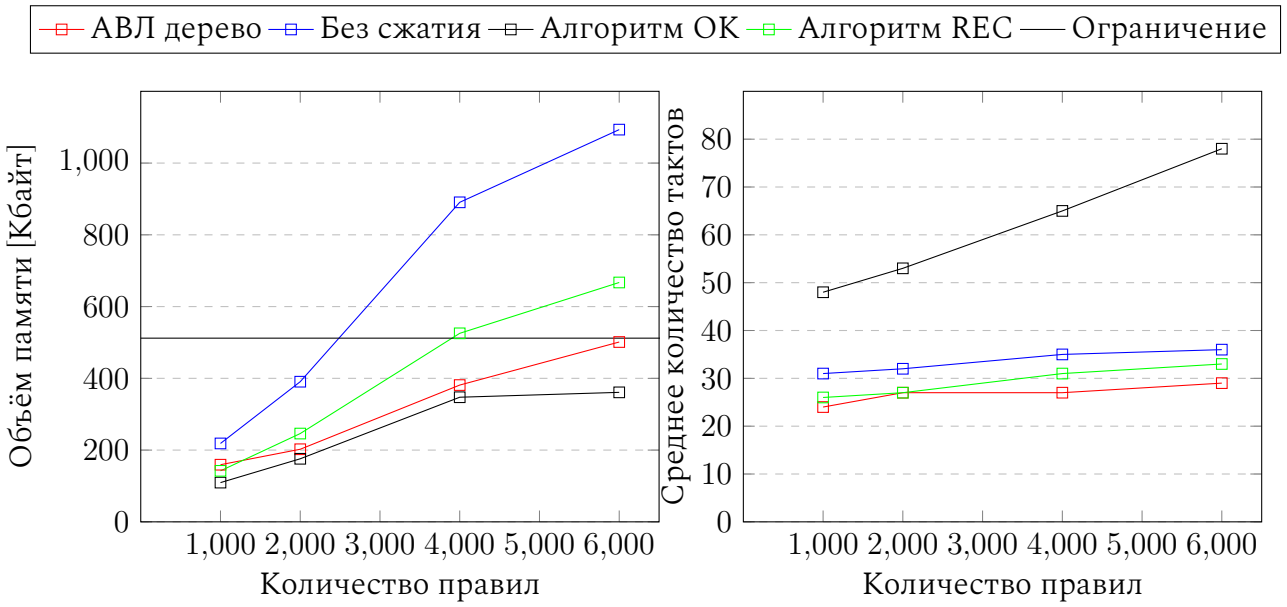
Рис. 6: Шаблон 2

или только значения $*$, что уменьшает размер программ полученных из соответствующих данным таблицам потоков. Для алгоритма рекурсивного отсечения получено уменьшение объёма памяти, используемого программой на языке ассемблера, в 1.5 раза. Для алгоритма оптимального кеширования получено уменьшение объёма памяти, используемого программой на языке ассемблера, в 2 раза.

Рассмотрим среднее количество инструкций, которые затрачиваются на обработку заголовка сетевого пакета в зависимости от количества правил и использованного алгоритма сжатия (рисунки 5b, 6b, 7b).

Так как алгоритм оптимального кеширования использует обращения к центральному процессору рассматриваемого коммутатора. На всех шаблонах при использовании алгоритма оптимального кеширования среднее количество затрачиваемых инструкций увеличилось в два раза.

Для остальных алгоритмов есть небольшое уменьшение среднего количества инструкций, так как сжатая таблица потоков содержит меньше правил, чем исходная.



(a) Зависимость затраченного объёма памяти от количества правил

(b) Зависимость количества тактов памяти от количества правил

Рис. 7: Шаблон 3

7.3 Выводы

По результатам экспериментального исследования были сделаны следующие выводы:

- Объем памяти, необходимый для хранения программы в сетевом процессоре, сокращается при использовании алгоритмов сжатия.
- Уменьшение количества признаков в правилах таблицы потоков не всегда приводит к уменьшению объема памяти, занимаемого соответствующей таблице программой. Большее влияние на размер имеет структура набора значений признаков правил.
- Использование алгоритма оптимального кеширования позволило сократить объём используемой памяти в $2 \ldots 2.5$ раза, что позволило использовать сетевой процессор для работы с таблицами большего объёма.
- Среднее количество инструкций, необходимое для обработки одного пакета, при использовании алгоритма оптимального кеширования значительно возрастает, но не выходит за границы.

8 Заключение

В рамках выпускной квалификационной работы разработаны алгоритмы сжатия таблиц потоков OpenFlow. Алгоритмы сжатия данных встраиваются в систему трансляции таблиц потоков в язык ассемблера сетевого процессора с использованием промежуточных структуры данных. В ходе работы были достигнуты следующие результаты:

- Проведён обзор существующих алгоритмов сжатия данных, в результате обзора были выбраны для реализации алгоритм оптимального кеширования и алгоритм рекуррентного отсечения;
- Выбранные алгоритмы сжатия были адаптированная для внедрения в транслятор таблиц потоков;
- Разработана программная реализация алгоритмов сжатия данных в трансляторе таблиц потоков;
- Проведено экспериментальное исследование разработанных алгоритмов сжатия, использования которых позволило снизить объём памяти, затрачиваемой на хранение таблиц потоков;
- Статья по результатам работы подана статья на конференцию SYRCoSE 2021;

Экспериментальное исследование показало, что применения алгоритмов сжатия позволяет сократить объём используемой памяти в несколько раз.

В качестве возможных направлений дальнейших исследований можно указать изучение возможности применения более продвинутых алгоритмов сжатия с использованием специализированной TCAM памяти.

Список литературы

[1] Wolfgang Braun и Michael Menth. “Wildcard compression of inter-domain routing tables for OpenFlow-based software-defined networking”. В: *2014 Third european workshop on software defined networks*. IEEE. 2014, с. 25—30.

[2] Yeim-Kuan Chang и Han-Chen Chen. “Fast packet classification using recursive endpoint-cutting and bucket compression on FPGA”. В: *The Computer Journal* 62.2 (2019), с. 198—214.

[3] *Development/LibpcapFileFormat - The Wireshark Wiki*. URL: https://wiki.wireshark.org/Development/LibpcapFileFormat.

[4] Open Networking Foundation. *OpenFlow Switch Specification Version 1.3.0 (Wire Protocol 0x04)*. 2012.

[5] Pankaj Gupta и Nick McKeown. “Packet classification using hierarchical intelligent cuttings”. В: *Hot Interconnects VII*. Т. 40. 1999.

[6] SR Kodituwakku и US Amarasinghe. “Comparison of lossless data compression algorithms for text data”. В: *Indian journal of computer science and engineering* 1.4 (2010), с. 416—425.

[7] A. Markoborodov, Y. Skobtsova и D. Volkanov. “Representation of the OpenFlow Switch Flow Table”. В: *2020 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*. 2020, с. 1—7. DOI: 10.1109/MoNeTeC49726.2020.9258208.

[8] N. Nikiforov, Y. Skobtsova и D. Volkanov. “Data Structures for Classification in the Network Processor without the Separated Associative Device”. В: *2020 International Scientific and Technical Conference Modern Computer Network Technologies (MoNeTeC)*. 2020, с. 1—2.

[9] Yaxuan Qi и др. “Packet Classification Algorithms: From Theory to Practice”. В: май 2009, с. 648 —656.

[10] Gábor Rétvári и др. “Compressing IP forwarding tables: Towards entropy bounds and beyond”. В: *ACM SIGCOMM Computer Communication Review* 43.4 (2013), с. 111—122.

[11] Ori Rottenstreich и János Tapolcai. “Optimal rule caching and lossy compression for longest prefix matching”. В: *IEEE/ACM Transactions on Networking* 25.2 (2016), с. 864—878.

[12] Zilin Shi и др. “MsBV: A Memory Compression Scheme for Bit-Vector-Based Classification Lookup Tables”. В: *IEEE Access* 8 (2020), с. 38673—38681.

[13] Sumeet Singh и др. “Packet classification using multidimensional cutting”. В: *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. 2003, с. 213—224.

[14] С. О. Беззубцев и др. “Об одном подходе к построению сетевого процессорного устройства”. В: *Моделирование и анализ информационных систем* 26.1 (2019), с. 39—62.

[15] Р. Л. Смелянский. “Программно-конфигурируемые сети”. В: *Открытые системы* 9 (2012), с. 15—26.